jamk.fi

I

Linux Patch Management: Comparison of Practical Implementations

Teemu Pulliainen

2016

Master's Degree Programme in Information Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

jamk.fi

Author	Type of publication	Date
Pulliainen, Teemu	Master's thesis	May 2016
		Language of publication: English
	Number of pages: 71	Permission for web publication: x

Title of publication

Linux Patch Management: Comparison of Practical Implementations

Degree programme

Master's Degree Programme in Information Technology

Supervisors

Rantonen, Mika

Häkkinen, Antti

Assigned by

Maatalouden Laskentakeskus Oy

Abstract

Recent public, high-profile vulnerabilities in widely used IT architecture components pose a significant risk to IT systems. As far as public vulnerabilities are concerned, In most cases a mitigation is readily available in a form of a software patch. The remaining issues are thus how to get informed about released patches, how to deploy them in a timely manner and how to ensure the patches are applied to all affected targets. The answer is usually some form of a patch management system.

Within a heterogeneous Linux Linux environment no established, industry standard way of deploying a patch management system exists. The objective was to study different deployment methods and select the most appropriate one according to the employer requirements, but also to give an insight on the others for the general public. An example implementation of three different patch management approaches was carried out in a test environment and each one researched and compared to the others. The first approach was creating a system from scratch the second using general-purpose configuration management software and the last was using a dedicated patch management software.

The research was successful. A functional implementation was accomplished with each approach and valuable experience was gained on each one. The positive and negative aspects of all alternatives were examined and recommendation for the overall best solution based on the employer requirements was also given. One should note, however, that such a recommendation is heavily dependent on the environment. The actual implementation to employer environment was not covered.

Keywords

Patch management, Configuration management, Linux

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

jamk.fi

Tekijä	Julkaisun laji	Päivämäärä
Pulliainen, Teemu	Opinnäytetyö, ylempi AMK	05 2015
	Sivumäärä	Julkaisun kieli englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Linux Patch Management: Comparing by Practical Implementations

Tutkinto-ohjelma
Master's Degree Programme in Information Technology

Työn ohjaajat
Rantonen, Mika
Häkkinen, Antti

Toimeksiantaja
Maatalouden Laskentakeskus Oy

Tiivistelmä
Viimeaikaiset laajaa julkisuutta saaneet haavoittuvuudet kriittisissä tietoteknisten järjestelmien komponenteissa aiheuttavat merkittävän riskin tietojärjestelmien toiminnalle. Yleiseen tietoon tulevien haavoittuvuuksien osalta korjauspäivitykset ovat useimmiten saatavilla. Jäljelle jäävät kysymykset koskevat siis sitä, kuinka saada tieto päivitysten julkaisemisesta, kuinka päivitykset asennetaan järkevässä ajassa ja kuinka varmistua päivitysten asentumisesta. Usein vastauksena toimii päivitysten hallintajärjestelmä.

Useita jakeluversioita käsittävien Linux-ympäristöjen osalta ei ole olemassa vakiintunutta tapaa ottaa käyttöön järjestelmä päivitysten hallintaa varten. Tavoitteena oli tutkia erilaisia tapoja hallintajärjestelmän käyttöönottoon ja valita niistä toimeksiantajan kriteereihin parhaiten sopiva toteutus. Toisena tavoitteena oli tarjota suurelle yleisölle tietoa eri tavoista päivitysten hallintaan Linux-ympäristöissä. Kolme erilaista esimerkkitoteutusta otettiin käyttöön testiympäristössä. Eri toteutustapoja tutkittiin ja verrattiin muihin. Ensimmäisenä toteutustapana toimi täysin itse kehitetty järjestelmä, toisena yleisten asetusten-hallintatyökalujen käyttöön perustuva järjestelmä ja kolmantena nimenomaan päivitysten hallintaan tarkoitettu järjestelmä.

Tutkimus onnistui ja jokaisella tutkitulla toteutusvaihtoehdolla saatiin otettua käyttöön mekanismi päivitysten hallintaan. Vaihtoehdoista saatiin kerättyä hyödyllistä tietoa, niiden hyviä ja huonoja puolia punnittiin ja toimeksiantajan vaatimusten perusteella saatiin aikaan suositus toteutettavasta järjestelmästä. Myös yleistä tietoa ja suosituksia eri järjestelmien soveltuvuudesta erilaisiin ympäristöihin saatiin tuotettua.

Avainsanat (asiasanat)

Linux, muutoksenhallinta, konfiguraation hallinta, tietoturvapäivitys

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Contents

Figures

Tables

Abbreviations

apt	Advanced Package Tool
CLI	Command Line
CSS	Cascading Style Sheets
CentOS	Community enterprise Operating System
DNS	Domain Name System
DoS	Denial of Service
EPEL	Extra Packages for Enterprise Linux
EPEL	Extra Packages for Enterprise Linux
ESXi	Elastic Sky X integrated
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTPS	Hypertext Transfer Protocol Secure
IDS's	Intrusion Detection Systems
IP	Internet Protocol
ITIL	IT Infrastructure Library
MAC	Media Control Address
MLOY	Maatalouden Laskentakeskus Oy
NAC	Network Access Control
NIST	National Institute of Standards and Technology
OSVDB	Open Source Vulnerability Database
PHP	PHP Hypertext Preprocessor
PIP	Pip Installs Python
PKI	Public Key Infrastructure
PSSH	parallel-ssh
RSA	Rivest, Shamir, Adleman
SSH	Secure SHell
SSL/TLS	Secure Sockets Layer/Transport Layer Security
STDERR	Standard error
STDOUT	Standard output
sudo	switch user do
tty	Teletype
UI	User Interface
URL	Uniform Resource Locator
VPN	Virtual Private Network
XML-RPC	eXtensible Markup Language – Remote Procedure Call
YAML	YAML Ain't Markup Language
yum	yellowdog updater modified

1 Introduction

1.1 Assigner

The thesis assigner Maatalouden Laskentakeskus Oy (MLOY, Agricultural Data Processing Centre Ltd.) is a leading software and IT solutions provider in the Finnish agricultural sector. Located in Vantaa, MLOY employs approximately 110 people and has a revenue of over EUR 9 million as of 2014. MLOY as a company was founded in 1986 but has history of over 60 years and now possesses a rare combination of both technological and agricultural competence. (Maatalouden Laskentakeskus Oy, 2014, 2-3).

1.2 Motivation and Background

The assigner already had a patch management policy and certain systems and guidelines in place to carry out the policy. Separate development, test and production environments did exist as well. However, the process was very Windows-centric and the process only partially covered the growing number of Linux hosts. Linux hosts in MLOY were updated manually, one by one as needed. The hosts were updated directly from their distribution repository servers over the Internet. The hosts might not be on the same patch levels and there was no clear picture what the current overall Linux patch situation was.

The primary motivator for the research was the requirement for a more streamlined and formal Linux patch management process by both the assigner and other technical personnel. The recent vulnerabilities with large scale and high visibility (Chapter 2) motivated the research even further. It was decided that the patching process should

ideally be technically trivial to follow and give the company an insight on the current patch situation.

1.3 Thesis Execution Phases

A set of requirements the resulting system should fulfill were assessed based on both company requirements introduced later and industry best practices described in the second chapter. The following three different approaches to implementing a patch management system were experimented with:

1. a self-created automatizing solution
2. implementation with existing automatizing tools and
3. dedicated patch management software.

A test environment was built and an implementation of each patch management system approach was carried out in the test environment. The capabilities of the implemented solutions options were evaluated based on the aforementioned requirements as well as existing deployments and literature. The different solutions were then compared with each other. Based on the requirements, the most suitable solution for the MLOY environment was selected for implementation in the production environment.

1.4 Objectives and Methods of the Research

The primary goal of this thesis was to review and compare different methods of implementing a patch management system. The review and comparison was done in or-

der to provide the assigning organization with a solid base for choosing and implementing such a system. Based on the author's familiarity with the environments as well as opinions by colleagues, the system was to ideally meet at least most of the following requirements for its features:

1. Comprehensive: include most of the Linux hosts within the organization, both Debian (6, “Squeeze”; 7, “Wheezy”; 8, “Jessie”) and Red Hat (6 and 7) -based ones.
2. Easy to use: usable for systems administrators with little Linux know-how, for example by utilizing a web front-end.
3. Controllable: include a mechanism to select which patches to apply and to which hosts.
4. Report-friendly: include a mechanism to show which patches have been applied, which are missing and a graphical summary.

Another objective was to provide the general public with an overview of different patch management solutions for a small-scale, but possibly complex and heterogeneous Linux infrastructure. The solution was to, however, ideally scale to more complex environments as well.

The primary research problem can be described as “*How to select the best patch management system*”. Due to the broad scope and the relative vagueness of the research problem, case study was selected as the research method. A case study is a “method used to narrow down a very broad field of research into one easily researchable topic” (Shuttleworth, 2008, 1). A case study also does not attempt to find an ultimate truth based on bare numeric data. Instead, the focus of a case study is to seek a holistic understanding of the situation and to provide “new variables and questions for further research” (Becker, 2012). The thesis includes mostly qualitative

elements (verbal capability assessment) but there are to be some quantitative ones (calculated, score-based capability assessment) as well. A hypothesis of the capabilities of each solution was made based on literature. A real-life experiment with the solution was then conducted to observe how it compared to the results achieved in real life.

1.5 Research Scope, Delimitation and Known Challenges

The thesis is supposed to give an overview of different patch management methods and offer a closer look on a few solutions. Based on time and environmental restrictions, the following delimitations were made:

- Since the size of the assigner's Linux infrastructure is limited, the focus will be on a relatively small-scale deployment.
- The solution is supposed to cover multiple operating systems, versions and architectures: however, based on the existing assigner architecture, only Debian 6, Debian 7 and Red Hat 6 -based operating systems on Intel x86 and x86-64 architectures are covered in the test implementation phase.
- The focus will be on systems operating on server roles, little thought will be given to desktop-related questions.
- Since configuration management is a wide topic of its own and there are numerous quite complex solutions to implement it, the configuration management focus will be on literature review, although one configuration management tool will be investigated in more detail.
- The thesis focuses primarily on patching the operating system: the kernel, userspace tools (bash etc.) and frameworks/middleware (Tomcat, PHP, MySQL etc.). Application patching will be given little attention.

- Different types of patches exist: they may provide functionality updates or new features as well as security updates (Nicastro, 2011, Chapter 1). The focus in this thesis is on security-related matters.

While automating a patch management process can yield substantial benefits, quite a few challenges were recognized already before the implementation phase. The question whether to address the following issues was postponed to the implementation phase:

- Change management: how to stay aware of installed software.
- How to get notified about available updates.
- How to keep track of patch levels between hosts and be certain a specific patch has been applied to all hosts required.

2 Background to the Research

The constant growth of threats against information systems in recent years has lead to a growing need to address public security vulnerabilities throughout the organization without delay. While it is indeed possible to update – or *patch* – each computer system individually, it is becoming more and more evident that a centralized updating system would be a more reliable, simpler and less time consuming option. Many organizations have already deployed a patch management system for their Windows infrastructure. Recent threats in Linux environments at the least – such as the OpenSSL vulnerability Heartbleed (Codenomicon, 2014), Bash shell vulnerability Shellshock (NVD, 2014) and C-library vulnerability Ghost (Sarwate, 2015) - have shown that there is also a growing demand for similar systems for Linux environments.

2.1 Vulnerabilities and Attack Vectors

The Heartbleed vulnerability, for instance, is a serious vulnerability in the ubiquitous OpenSSL cryptographic library which often provides cryptographic functions for web, email, VPN (Virtual Private Network) and instant messaging services. When disclosed in April 2014 it ended up in news headlines and created a strong pressure for the technical staff to address the issue without delay (Aro, 2014). What made the Heartbleed vulnerability particularly notorious was the fact that by exploiting it, private information – such as private cryptographic keys – could potentially be disclosed without leaving a trace. This led to a massive number of SSL/TLS (Secure Sockets Layer/Transport Layer Security) certificates being revoked and reissued. Another particularly nasty feature of the Heartbleed bug was that it resided in many devices that were not updated regularly if ever. (Codenomicon, 2014). Possible attack vectors for the Heartbleed vulnerability include a public Apache web server over HTTPS (Hypertext Transfer Protocol Secure) using vulnerable OpenSSL libraries.

The GHOST vulnerability – named after the vulnerable *gethostbyname()* function – is a serious vulnerability in the glibc library. Glibc is an implementation of the standard C library often used as a core component in Linux environments. GHOST is a buffer overflow vulnerability in a glibc function providing name resolution that can be triggered remotely and could lead to a total system compromise. Example exploits against the Exim email server were quickly published after the vulnerability was disclosed in January 2015, followed by a ready-to-use Metasploit module. Potentially, any software component that can be deceived to make a DNS (Domain Name System) resolution query by utilizing the gethostbyname() function in a vulnerable glibc installation could be used as an attack vector. Like Heartbleed, the vulnerability was extremely widespread due to the popularity of the glibc library. (Sarwate, 2015).

Another example is a vulnerability in the ImageMagick image manipulating library, labeled ImageTragick. The ImageTragick bug could allow remote code execution for instance on a web forum software. Remote code execution could be triggered if a user is able to upload a malicious image which is then processed by the vulnerable software component. An example of the above is a public WordPress site which allows public image upload and uses ImageMagick software or the PHP (PHP Hypertext Preprocessor) extension php-imagick for image manipulation. When disclosed on May 4th 2016, only a workaround existed, as well as partial fix on the ImageMagick source code. Binary packages offered by numerous operating systems were still vulnerable with updates expected on the coming days. (Ermishkin, 2016).

On the previous examples a patch management system – along with a patch management process – would greatly improve the patch coverage (which platforms are already patched). A patch management system could also help in determining when a patch is published and how severe the addressed vulnerability is. As Nicastro (2011, Chapter 1) puts it, ”*A comprehensive security patch management process is a fundamental security requirement for any organization that uses computers, networks, or applications for doing business today*.”

2.2 Theory and Processes of Patch Management

Soyppaua (2013, 2) defines patch managements as “*the process for identifying, acquiring, installing, and verifying patches for products and systems*”. Patch management is often considered a subset of configuration management and closely related to risk management as well. When implementing a systematic patch management approach, it is usually required to analyze the *threat* and the *vulnerability* that the patch applies to in order to be able to calculate the *risk* of not patching. (Voldal, 2003, 1-2).

Usually multiple, perhaps overlapping, mechanisms for applying patches exist. Such mechanisms could include the following (Soyppaua, 2013, 4):

- Self-updating software
- Operating-system level centralized update tool [such as yum (yellowdog updater modified)]
- Third-party management application (such as Spacewalk)
- Network Access Control (NAC) software
- User-initiated installation or upgrade.

Developing or implementing a patch management system should most likely begin by implementing a patch management process (Nicastro, Chapter 3). The process itself often defined as a flow consisting of different key stages. Nicastro (Chapter 4) suggests utilizing ITIL (IT Infrastructure Library) to creating such a process.

In Nicastro's approach, a patch management process consists of the seven stages illustrated below in Figure 1:

Figure 1. Architecture proposal (Nicastro, 2011, Chapter 3)

White (2004, 3-5) takes quite a similar approach defining his patch management process in the following seven stages:

Information Gathering

White further divides the first stage, information gathering, into two sub-stages, *asset and host management* and *vulnerability notification*. Asset and host management phase includes identifying which hosts and what kind of software is running on the network. White suggests a continuous process of identifying these elements and a *passive* method where no agent software is required. Collected information should include IP (Internet Protocol) and MAC (Media Control Address) addresses and software version information as well as server functions and running services. In order to be able to determine whether a patch should be installed immediately or not, systems should then be classified according to levels of importance. White suggests three levels: *mission critical* (highest, critical to business operation), *business critical*

and *operational critical* (lowest, can tolerate service breaks). Similarily, Nicastro (Chapter 3) finds an accurate inventory of systems essential in order to implement different patching strategies for different kinds of systems: desktops, servers and network appliances.

In White's work (2004, 4), vulnerability notification sub-stage relates to gathering information about found vulnerabilities and released patches. The information should include affected software versions, seriousness rating and possible temporary workarounds. White suggests subscribing to relevant security announcement lists and public disclosure lists to achieve this information.

In a similar fashion, Nicastro (2011, Chapter 2) suggests one of two approaches to tracking new patch releases: either crawling through known-good web page sources and subscribing to relevant mailing lists, or using a third-party service or software tool. If using the first option, Nicastro lists a few information sources such as United States Computer Emergency Readiness Team (US-CERT) ”*Technical Cyber Security Alerts*” and ”*@Risk: The Consensus Security Alert*” by SANS Institute. When using the latter option, Nicastro suggests handing over the systems inventory to this third party in order to receive more relevant information. In chapter 5, Nicastro points out that in addition to external vulnerability sources, internal ones should be used as well. These include information gathered from network devices, IDSs (Intrusion Detection Systems), firewalls and so on as a part of routine security operations. This information could help in finding vulnerabilities as well as assessing the threat to the particular systems.

Scheduling

White (2004, 4) proposes scheduling patches in two different cycles. First, there should be a normal cycle to install non-critical patches e.g. monthly or whenever service packs are released. In addition, critical patches should be installed whenever they are announced. Patch priority should be calculated based on *criticality* an-

nounced by the vendor, *availability* of exploit, *importance* and *exposure* of the affected systems and exposure of the affected systems. Nicastro (2011, Chapter 9) also suggests using static time frames for patch installation based on the severity of the vulnerability. In Nicastro's example a time frame for emergency-level patches could be within twelve hours whereas information-level patches might be left uninstalled altogether.

Testing

Testing should be considered critical to the patch management process and should be done in an exact mirror of the production environment whenever possible. If this is not possible, the patches should first be rolled out to easily-recoverable systems. (White, 2004, 4-5). Nicastro (2011, chapter 8) recognizes the same issues and also recommends using a patch rating system to be able to concentrate on the patches that have the biggest effect on the organization. A virtualized test environment could be created as well as recommended by both White (2004, 4-5) and Nicastro (2011, Chapter 8).

Change Management

Change management stage includes planning for patch implementation. White (2004, 5) suggests contingency and back-out plans should first exist. Documentation about what is installed and how to remove it should exist, as well as backups and standby personnel in case of failure. Risk mitigation should be considered as well so that possible complications are limited. Plans should exist for defining what criteria must be met for the patch to be considered successful.

Installation

White (2004, 5) calls for a well controlled and predictable installation to limit disruption to business services. Access controls should be implemented to disallow out-of-

band patch installation by uses or automated processes. Guidelines should be written and checked regularly. Mission critical systems should be patched manually, during off-hours. Just as White, Nicastro (2011, Chapter 8) mentions the need for a well-controlled implementation by the operations group as well. Nicastro pays special attention to using pre-generated step-by-step instructions in the implementation phase.

Audit and Assessment

Successful patch deployment should be verified by using e.g. agent software or vulnerability scanner. Reports should also be generated at this stage as well as documentation about any occurred issues. The reports should be forwarded to upper management so that they know the process is functioning as expected. In Nicastro's approach (2011, Chapter 9) a named security group generates reports and maintains them in a central repository or database. The documentation should include an overview of the vulnerability, test plan and results, implementation and back-out plans for all systems as well as progress reports and scorecards to track patched systems.

Consistency and Compliance

White (2004, 5) explains this stage as a kind of a feedback where the lessons learned are used to improve the process. An audit trail is also provided in this stage. The real-world product of this stage could be an update of images, build scripts and documentation.

2.3 Architecture of Patch Management Software

There are numerous architectural options for implementing a patch management system. Souppaya (2013, 8), for instance, states that a patch management system consists of "*one or more centralized servers that provide management and reporting, and one or more consoles*". Souppaya lists three different methods of identifying missing patches (2013, 8-9):

- agent-based,
- agentless scanning, and
- passive network monitoring.

In the agent-based model a separate agent software is installed on target hosts. The agent communicates with a separate, centralized management server. The agent is responsible for determining which vulnerable software is present on the host, updating it and executing any state required changes such as restarting the host. Soyppaya (2013, 8) recommends agent-based architecture for mobile hosts such as telecommuter laptops and smartphones. Souppaya finds limitations for this architecture as well: agents usually cannot be installed on locked appliances and agents may not be available for all operating systems. (Souppaya, 2013, 8-10.)

As opposed to agent-based scanning, in agentless scanning architecture there is only a scanner server scanning the hosts for vulnerable software. The scanning server usually requires root level access to the hosts. On the positive side, this method does not require agent installation. The downside is that the hosts need to be in the local network. Network security devices may also block the scan attempts, and scanning might consume bandwidth.

Finally, in passive network monitoring, vulnerable software is identified from network traffic. No privileges are required on the hosts so this method could be used e.g. against third-party devices where the company has no access. Limitations include limited visibility (applies only to such situations when e.g. software version can be identified from the traffic it generates). This method is also only available in the local network (Souppaya, 2014, 8-10.)

White (2004, 7-8) categorizes the architectural classes in a similar fashion. White's *Client-Server* architecture is analogous to Soyppaya's (2013, 8) agent-based scanning and *server-only* architecture closely resembles Souppaya's agentless scanning. White also lists the use of *Configuration Managers* – such as IBM Tivoli – as a category of its own. White defines Configurations Manager category as agent-based systems where patch management is only a subset of their full functionality.

In addition to the above, Nicastro (Chapter 3) mentions remote user devices as an area to give special attention to. While agent-based approach could be used, the agents and end user devices are less trusted in this scenario. As an example, the use of a separate, isolated network segment is suggested so that VPN connections to the local network are confined within the isolated segment until they are patched or otherwise found eligible for full network access.

White (2004, 8-10) also proposes an architecture of an automated, open source, cross platform patch management system based on his patch management process. White's approach is holistic aiming to provide a complete system covering detection, testing and deployment as well as reporting and maintenance. In White's opinion, the system should meet the following criteria in addition to the obvious patch installation functionalities:

- be secure,
- be inexpensive,
- be vendor-neutral
- have powerful feedback mechanisms
- be flexible to ensure smooth integration with e.g. IDS's (Intrusion Detection Systems)
- be robust and easy to use

White (2004, 8-10) describes a number of key elements a patch management system should include closely related to his patch management process. A depiction of White's architectural proposal is illustrated in Figure 2.

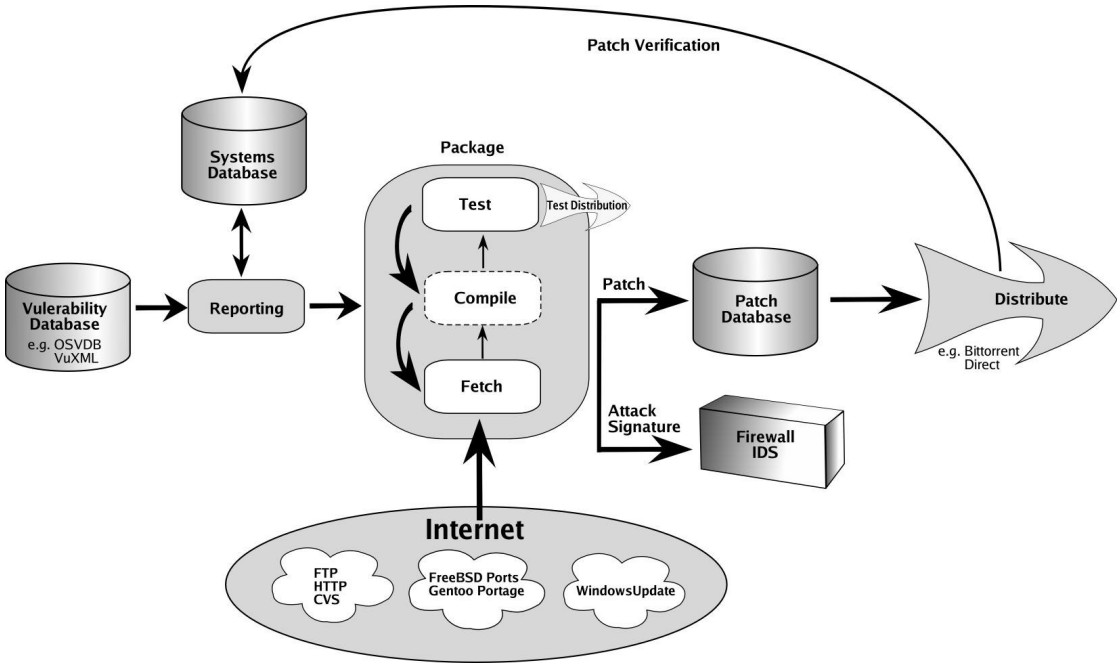

Figure 2. Architecture proposal from White (2004, 2)

Vulnerability Notification

In the vulnerability notification phase a current and dynamic database of vulnerabilities is required. White (2004, 8) prefers an externally maintained database due to the high volume and broad spectrum of vulnerabilities. White proposes the Open Source Vulnerability Database (OSVDB) due to its size and the XML-RPC (eXtensible Markup Language – Remote Procedure Call) interface provided. It should however be taken into account that a decision to shut down OSVDB was published on April 2016 (OSVDB, 2016).

Reporting

Patches need to be thoroughly tested before installing in production and processes for installing and removing patches need to be documented as well, which can potentially be very time-consuming. White (2004, 9) proposes a systems database keeping track of host names, locations, IP addresses, software versions etc. White also proposes using a vulnerability scanner such as Nessus to ensure patch installation.

Pulling Patches

Patches are usually not distributed from a single location, and there could be multiple vendors, each with their own patch distribution policy. White (2004, 9) suggests a modular approach where a different fetching method is implemented as a plugin for each source.

Creating Packages

White (2004, 9) presents two different methods of package creation: replacing the entire binary or using binary patching. Binary patching reduces distribution time; however, is more likely to raise unexpected issues due to varying system configurations. White suggests a separate, shared patch database – possibly maintained by an industry body – so that different organizations could learn from each other's patching techniques.

Testing Packages and Stop-Gap Protection

White (2004, 9-10) explains the difficulty of automating testing and concludes that it will most likely be a time-consuming, manual task. During this time, the production systems could be protected by using IDS signatures to drop malicious traffic. White proposes using the vulnerability information in OSVDB to create IDS signatures as a long-term plan.

Patch Distribution

White (2004, 10) states that single-server patch deployment is inefficient and prone to DoS (Denial of Service) attacks. He proposes using peer-to-peer distribution instead, such as Bittorrent. This would reduce bandwidth load on the distribution server and provide greater security.

2.4 Introduction to Linux Software Management

There are numerous different ways to install and update software in Linux environments, such as building from source code or using vendor-specific binary installers much alike in Windows infrastructure. The most common way, however, is to utilize distribution-specific package management system such as *yum* in Red Hat Enterprise Linux and its derivatives or *apt* (Advanced Package Tool) in Debian and its derivatives. Package management systems handle software installation, its dependencies such as specific libraries and sometimes configuration as well. The package managers use binary software packages pre-compiled for the distribution. The packages in turn reside in archives called *software repositories*. Multiple different repositories can be used in a single operating system instance, allowing the use of self-hosted repositories with possibly custom software.

3 Test Environment

3.1 Overview

A test environment was created in a VMware ESXi (Elastic Sky X integrated) virtual environment consisting of the following virtual machines:

- Manager: the patch management host
- Centosrepo: a local software repository mirror
- Centos6: a test host running CentOS 6
- Debian6: a test host running Debian 6, “Squeeze”
- Debian7: a test host running Debian 7, “Wheezy”
- Debian8: a test host running Debian 8, “Jessie”.

Using a virtual machine environment made it possible to return to specific point in time configuration-wise: for example, reverting a host to a point before installing a specific piece of software.

3.2 Internal Software Repository

An internal software repository was implemented on the Centosrepo host. The purpose of this repository was to

- control the updated software versions

- be able to isolate the hosts so that they are not able to connect to the Internet
- avoid bottlenecks caused by remote repositories by downloading patches over a fast local network and
- save bandwidth by only downloading from over the Internet once.

By implementing a local repository, it was possible to limit Internet access to the repository host only as depicted in Figure 3.

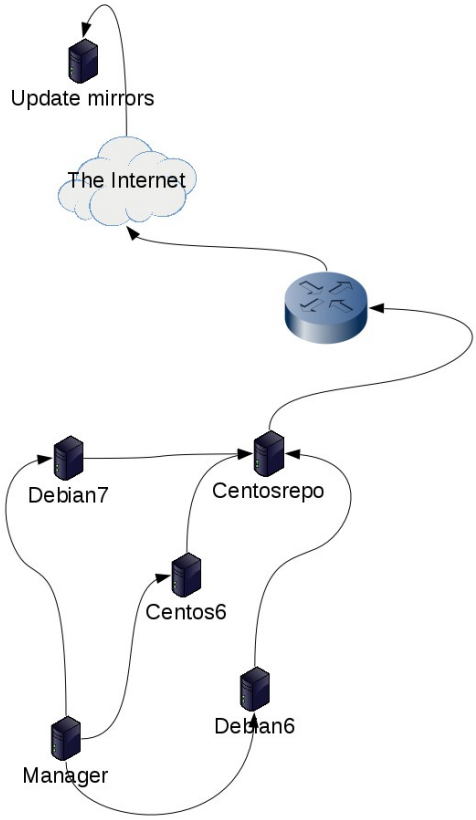

Figure. 3. Test network topology.

4 Implementation Options

4.1 Self-created

The basic idea in a self-created patching system was to automatically log in to the hosts and utilize the existing operating system update functions, such as yum on CentOS (Community enterprise Operating System) hosts and apt on Debian hosts. Compared to the NIST (National Institute of Standards and Technology) guide (Souppaya, 2013, 8-9) this method includes features from both agentless network scanning and agent-based scanning. No distinct agent is installed, but package management features native to the operating system are used as an agent.

4.1.1 Preparation

Since SSH (Secure Shell) is the de facto standard way of logging on remote Linux machines and multiple simultaneous logins were required, PSSH (parallel-ssh) tool was chosen as the login tool. PSSH is a Python application providing parallel versions of OpenSSH and related tools (McNabb, 2015). PSSH does not require a GUI (Graphical User Interface) and is well suited for plain shell environment. PSSH was installed on the Manager host by enabling the Extra Packages for Enterprise Linux (EPEL) repository and installing PSSH from the repository.

SSH logins needed to authenticate against the remote hosts without prompting for credentials. A regular user 'patch' was created on all hosts. In order to avoid root login the user was granted sudo (switch user do) permissions to run commands by adding the following definitions in the *sudoers* file:

```
patch   ALL=(ALL:ALL) NOPASSWD:ALL
```

As a security best practice, the sudo permissions should be far more limited. However, the update scripts regularly need to restart services etc. so simply allowing only *apt-get* and *yum* commands would not be enough.

A cryptographic RSA (Rivest, Shamir, Adleman) key pair was created by using the ssh-keygen utility. No passphrase was set in order to allow unattended authentication. The private key was stored on the Manager host and the public key was distributed to authorized_keys file on the .ssh directory under 'patch' user's home directory on Centos6, Debian6 and Debian7 hosts by using the ssh-copy-id utility.

4.1.2 Testing Patch Automation on Command Line

Parallel-SSH was used to execute update commands on hosts. First, hosts were divided into groups. Groups were based on whether the exact same commands could be executed on members of a group. Different files were created per server groups so that CentOS-based server Centos6 was on one group and Debian6 and Debian7 hosts were on another. group The syntax of the files consisted simply of the host name or IP (Internet Protocol) address and username as follows:

```
patch@172.17.1.101
patch@172.17.1.103
```

Unattended login for user patch was already set up, so at this point it was possible to run commands simultaneously on multiple hosts. PSSH usage was tested by using the hosts file as list of target hosts and “touch testfile” as the command to be remotely executed. An '-i' parameter was provided for PSSH to direct STDOUT (standard output) and STDERR (standard error) streams from the target hosts to the shell running

PSSH. Similarly, an 'ls' command was executed to verify that the touch command had indeed created empty files on the target hosts:

```
[root@Manager ~]# pssh -h debian_hosts -i touch testfile
[1] 16:58:06 [SUCCESS] patch@172.17.1.103
[2] 16:58:06 [SUCCESS] patch@172.17.1.101
[root@Manager ~]# pssh -h debian_hosts -i ls
[1] 16:58:10 [SUCCESS] patch@172.17.1.101
testfile
[2] 16:58:10 [SUCCESS] patch@172.17.1.103
testfile
```

4.1.3 Patch automation on CLI: an interactive approach

While creating the parallel-SSH solution, it was noticed that in certain environments an interactive approach might be more preferable. ClusterSSH was used for this purpose. ClusterSSH is a Perl-based wrapper utilizing standard Linux tools such as ssh and xterm. It is originally intended for cluster administration, however, since it essentially only offers a single interface to commit commands to multiple target hosts, it can easily be used for other purposes as well. (Childers, 2011.)

Since ClusterSSH requires a Graphical User Interface, it was installed on an external desktop host from the distribution repository. After a PKI (Public Key Infrastructure) authentication scheme was created in a similar fashion to the previous chapter, ClusterSSH could already be used in its simplest form with the following command:

```
# clusterssh patch@debian6 patch@debian7 patch@centos6
```

The command above effectively opened a control window and a terminal window per each connected host. Since the point of clusterssh is to pass identical commands to multiple hosts, the hosts were grouped to ClusterSSH clusters. The division was made per operating system so that Debian6 and Debian7 were on cluster “DebianCluster” and Centos6 was on “CentosCluster”. In addition, a meta-cluster “All” grouping both clusters was created. The clusters were defined in the ~/.clusterssh/clusters configuration file in the following way:

```
DebianCluster=Debian6 Debian7

CentosCluster=Centos6

All=DebianCluster CentosCluster
```

It was now possible to connect to Debian hosts by issuing command *clusterssh -l patch DebianCluster* or to all hosts by command *clusterssh -l patch All.* Commands could be run simultaneously on the cluster by typing into the CSSH command window as demonstrated on Figures 4 and 5.

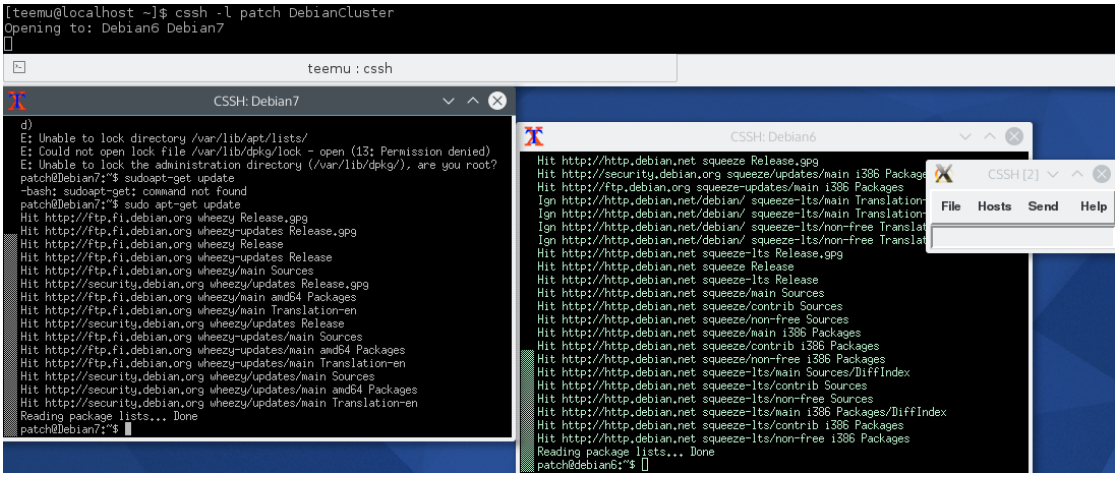

Figure 4. Running apt-get on Debian cluster

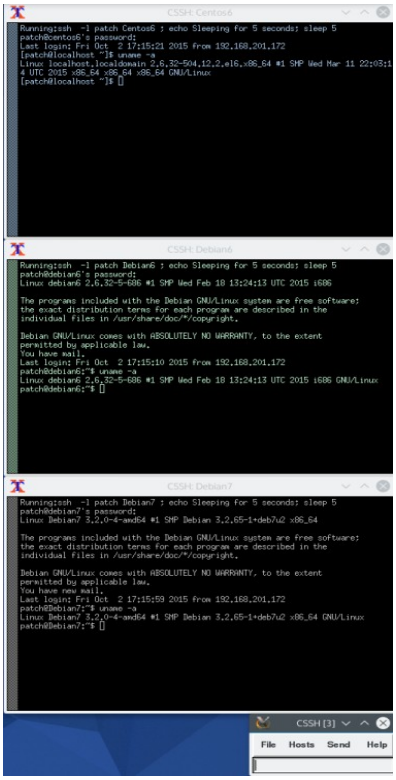

Figure 5. Running uname on all clusters.

4.1.4 Creating a Proof-of-Concept Web Interface

A simple interface was also required to allow people with little shell experience to execute the patching commands. Since a Python environment is usually installed on Linux hosts and was already running on all hosts, the Python Django web framework was used to build a web user interface.

The Manager system was used to host the web interface. Python and the Django Framework were provided on CentOS 6 base and EPEL (Extra Packages for Enterprise Linux) repositories, however, the versions were fairly old. In order to get the more recent versions, the CentOS Software Collections was used to install Python. The PIP (Pip Installs Python) package management system was installed as well and in turn used to install the latest Django framework. The end result was Python 2.7 environment with the latest Django version 1.7.

A new Django project “Patching” was created to host the application and the new application “PatchWeb” was created. The following django “view” - or function – “command_view” was created in the PatchWeb application:

```
def command_view(request):
    output = os.popen('pssh -h /root/debian_hosts -i ls').read()
    return render(request, 'command.html', {'output': output})
```

The function utilizes the Python “os” library, which enables running operating system commands in Python applications. Here, the previously tested pssh-command was defined as the “output” variable and the output of the command was used as the re-

turn value of the function. An extremely simple HTML (Hypertext Markup Language) template “command.html” was created to handle the output:

```
<html>
<h1> Output of our command:</h1>
<pre> {{ output }}</pre>
</html>
```

The output of the command_view function was called on line 3.

Finally, the urls.py file of the Patching project was modified so that when entering the root URL (Uniform Resource Locator) of the application, the command_view function would be called:

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
from PatchWeb.views import command_view

urlpatterns = patterns('',
   url(r'^admin/', include(admin.site.urls)),
   url(r'^$', command_view),
)
```

At this point there was a function to execute the required task, a handle to call the function and an HTML page to show the output on. To demonstrate the functionality,

the Django built-in web server was started to serve the application and a browser was pointed to the web server socket on the Manager host. As expected, the same PSSH command output as before was shown but this time on the browser window (Figure 6).

Figure 6. Initial PSSH output on browser

4.1.5 Creating the Patching Application

A patching application was created based on the proof-of-concept terminal-based patching scripts and web automation on previous chapters. The goal was to provide a web interface with a possibility to view available patches per host and apply them. Standard operating system level update procedures were supposed to be run on different operating systems. Since these procedures differ per operating system, an additional file centos_hosts – consisting of CentOS host IP addresses – was created in addition to the previously created debian_hosts file.

At this point the basic idea was to run the required *yum* and *apt* commands in a similar manner to the *ls* command previously, simply like *pssh -h debian_hosts -i apt-get*

dist-upgrade. As the installation was supposed to be unattended and run via an SSH connection, the following additional issues were, however, identified:

1. Starting the update requires user confirmation,
2. Additional user input might be required during the update,
3. *sudo* requires a *tty* (Teletype, that is, a terminal session) on CentOS hosts and is problematic even if provided one and
4. pssh has a default timeout of one minute so long-running update processes time out before completing.

To overcome the first limitation, parameters –assume-yes (apt) and –assumeyes (yum) were given to the remote command to answer “yes” to all prompts. It was also known that dpkg, the Debian package manager, would sometimes prompt for user input when a new configuration file was available. Dpkg itself provides a way to assume a certain answer. Since apt-get provides a -o switch to give instructions to the underlying dpkg system, the following parameter was given to apt-get:

```
-o Dpkg::Options::="--force-confold"
```

This way old, possibly self-modified versions of configuration files would always be kept without asking for user input (Hertzog, 2016, Chapter 6.8).

One more user input possibility was identified on Debian hosts while updating the wget package: the new version release notes were printed on screen and escaping the notification screen required user input. This was fixed by setting the environment

variable DEBIAN_FRONTEND as 'noninteractive' before running the apt-get command (Hertzog, 2016, Chapter 6.8).

The third issue, *sudo* requiring a TTY was initially fixed by passing the SSH connection a pseudo-TTY with the -x '-tt' parameter. While this did fix the issue with lacking TTY, the following error was often seen: *tcgetattr: Invalid argument.* It would appear that this is an issue needed to be addressed with pssh (Parallel-SSH Issue Tracker, 2013,. issue #90). As a workaround CentOS hosts settings were changed so that login was done directly as user root without using sudo.

Finally, a timeout of ten minutes was set for pssh connections with the -t switch. The hosts were successfully updated by issuing the following pssh commands on Manager host:

```
pssh -i -t 600 -h /root/centos_hosts yum --assumeyes update # upgrade CentOS

pssh -i -t 600 -h /root/debian_hosts  sudo apt-get update #update Debian package index

pssh -i -t 600 -h /root/debian_hosts sudo DEBIAN_FRONTEND=noninteractive apt-get
dist-upgrade -o Dpkg::Options::="--force-confold" --assume-yes #upgrade Debian
```

On the web application side, the command_view function was modified so that it would list all available hosts by printing the hosts files:

```
def command_view(request):
    output = os.popen('cat /root/debian_hosts; cat /root/centos_hosts').read()
    return render(request, 'index.html', {'output': output})
```

In addition, another function listUpdates was defined to run an update check. The purpose of this function was to run yum or apt update commands on the respective target machines via pssh and print the output. Parameters '-s upgrade' and 'check-update' were provided for apt and yum commands in order to only simulate the update, thus listing available updates:

```
def listUpdates(request):

    output = os.popen('pssh -i -h /root/centos_hosts yum check-update ; pssh -i -h
/root/debian_hosts apt-get -s upgrade').read()

    return render(request, 'listUpdates.html', {'output': output})
```

Yet another function runAllUpdates was also defined to execute the actual update commands and use the HTML template runAllUpdates.html for displaying them:

```
def runAllUpdates(request):

    output = os.popen('pssh -i -t 600 -h /root/debian_hosts  sudo apt-get update; pssh -i
-t 600 -h /root/centos_hosts yum --assumeyes update ; pssh -i -t 600 -h
/root/debian_hosts sudo DEBIAN_FRONTEND=noninteractive apt-get dist-upgrade -o
Dpkg::Options::="--force-confold" --assume-yes').read()

    return render(request, 'runAllUpdates.html', {'output': output})
```

The Patching Django project urls.py file was modified so that requesting URL /runAllUpdates.html would call the runAllUpdates function:

```
from PatchWeb.views import runAllUpdates

…

url(r'^runAllUpdates.html$', runAllUpdates),
```

A CSS (Cascading Style Sheets) file (Appendix 1) was created to simplify the web application layout management. In order to serve this CSS file – which from the framework's point of view is static content – STATIC_URL and STATICFILES_DIRS variables were set in the settigs.py file of the Patching project. The CSS file was created in the 'static' directory. In the CSS file different HTML DIV elements were defined to create the header on top of the application web page, a menu on the left and a space for output on the right.

HTML templates listUpdates.html and runAllUpdates.html were created (Appendix 2) as well to provide links for the urls in urls.py and to display the results. A javascript onclick confirm function was added to the runAllUpdates link to provide a confirmation window on the browser and prevent accidental updating.

The application was tested by first clicking on the 'Show available updates' link which displayed the result of the *apt-get* and *yum* simulation commands. Subsequently the 'Run all updates' link was clicked which first produces a confirmation box and then after a while the output of a successful update on all hosts. The result was confirmed by running the check procedure again: as expected, this time no updates were listed as being available since all patches had already been applied.

The web interface is presented below. Figure 7 presents the front page and host listing.

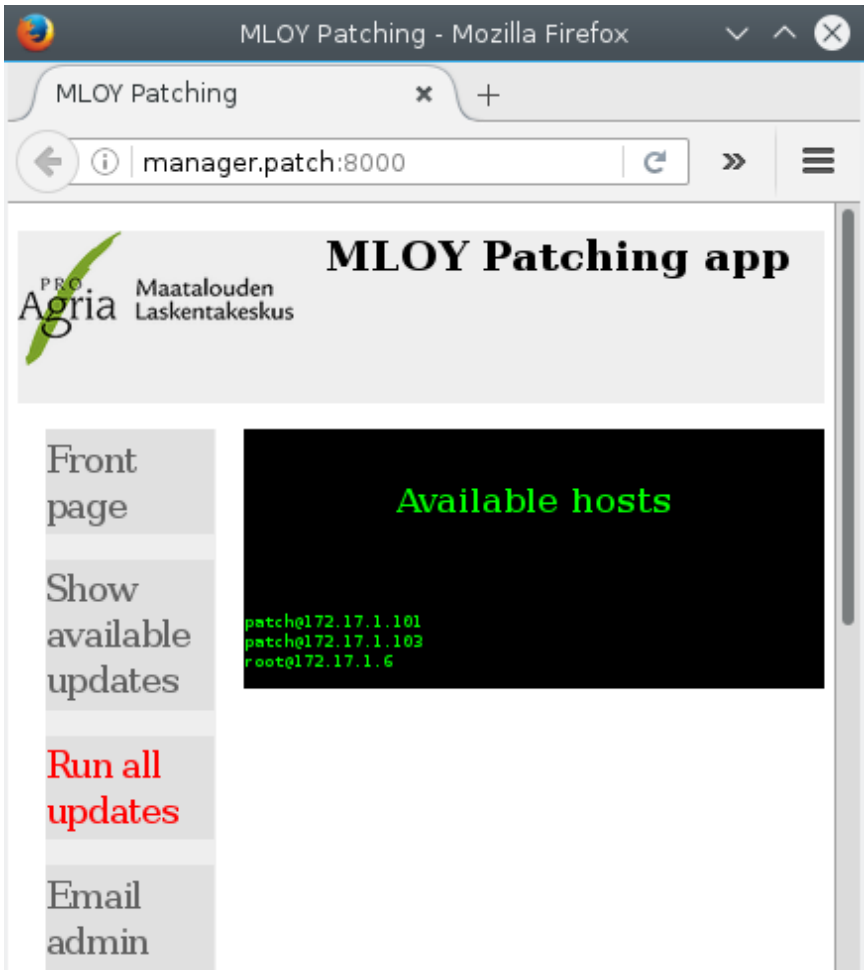

Figure 7. Front page with host listing.

The confirmation dialog, along with the available updates, is shown in Figure 8.

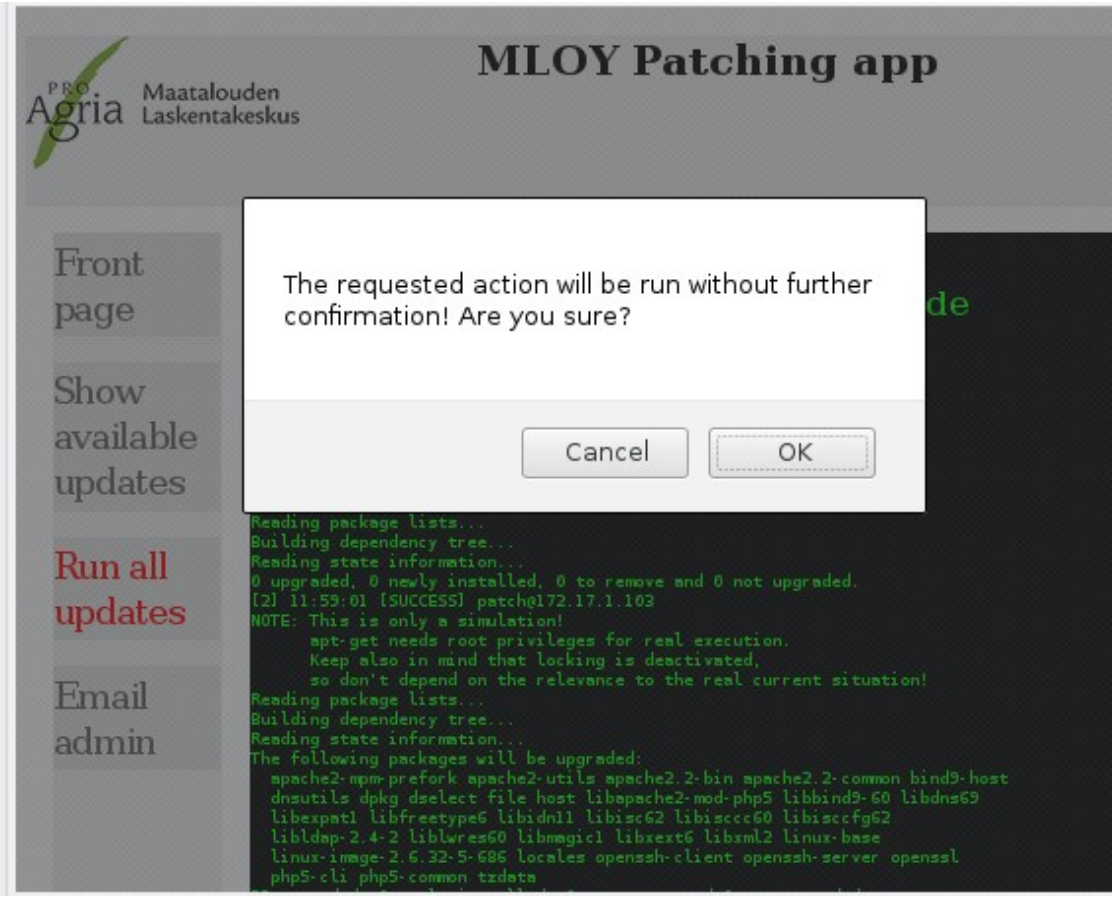

Figure 8. Confirmation dialog before update.

Finally, The actual update deployment process is shown in Figure 9.

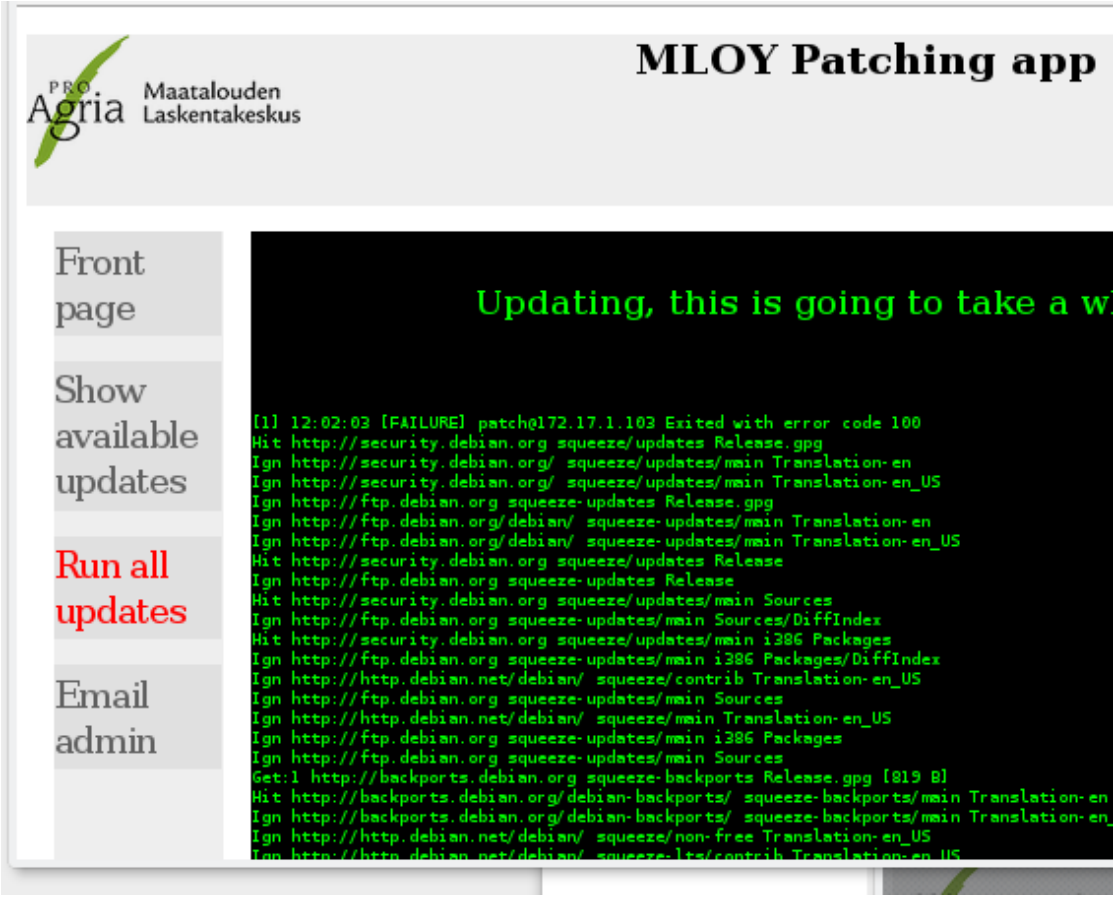

Figure 9. Deploying updates.

4.2 Automatizing with Configuration Management Software

Configuration management can be understood in several different ways. Jonassen Hass (2003, 3) describes it the following way:

"*Configuration management is unique identification, controlled storage, change control, and status reporting of selected intermediate work products, product components, and products during the life of a system.*"

The basic principles of configuration management systems include the following:

- making mass deployments easier and less time-consuming
- ensuring the configuration is identical across multiple environments.

Configuration management software offers a different approach to patch management. Whereas dedicated patch management applications offer a targeted feature set for just patching and a fully custom solution provides ultimate flexibility, a configuration management system sits in the middle ground. All sorts of configuration management tasks can be carried out while still being able to utilize the existing building blocks the management suite provides.

4.2.1 Introducing Configuration Management Systems

There are plenty of different configuration management applications for Linux systems, such as Puppet, Chef, CFEngine, SaltStack and Ansible. In the following chapters a few of them are briefly reviewed and one is selected for closer testing to assess whether it is suitable for patch management as well.

Puppet

Puppet, mainly developed by Puppet Labs, is arguably one of the oldest and most widely used configuration system software. Puppet comes in two flavors, the commercial Puppet Enterprise and freely available Open Source Puppet (Puppet Labs, 2016). Written in Ruby, Puppet can be run on most operating systems, such as different Linux distributions, BSD variants or Microsoft Windows. In a similar fashion to Ansible, Puppet uses a state-driven approach. Puppet is typically used in a client-server formation where the client is running agent software. The desired state is defined on the server – known as the Puppetmaster – and the clients periodically contact the server verifying their state matches the defined state. There are multiple GUI frontends available for Puppet. (Puppet Labs, 2016.)

Chef

Chef is a configuration management and automation platform from a company with the same name. Like Puppet, Chef is built on Ruby and also relies on it configuration-wise: Chef configurations – or Cookbooks – are written in Ruby. Chef is generally considered to be a more developer-oriented tool than Puppet or Ansible (Dreyfuss, 2015) and not as easy to get started with. In a similar fashion to Puppet, it utilizes a client-server approach with distinct agent software installed on the client hosts.

Ansible

According to Ansible Inc. (b) (2016), ”*Ansible is a radically simple IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and many other IT needs.*” The describing characteristics on Ansible are the following:

- Agentless: Ansible only uses SSH to communicate with the target hosts
- State-driven: Ansible defines the desired end state of the target, not the paths to get them to this state (Ansible Inc. (c) 2016). It is, however – possible to execute ad-hoc tasks as well.
- Smooth learning curve: Includes hundreds of existing modules, the host inventories are simple plaintext files and the automation language consists of simple YAML-formatted (YAML Ain't Markup Language) files called Playbooks.

An official graphical user interface, Ansibler Tower, is available as well as an open-source alternative, semaphore (Kramny, 2016). Ansible is built on Python.

4.2.2 Deploying Ansible

As per the previously set thesis delimitations, only Ansible was selected as the configuration management tool to further test automation with. The reasons for selecting Ansible included the expected relative ease of use, low infrastructure requirements and simple set-up.

Initial Ansible setup

Ansible with its dependencies was installed by building from source via PIP. The test hosts were defined in Ansible hosts file /etc/ansible/hosts. Authentication using SSH keys was already set up in Chapter 4.1. The initial setup was tested by issuing an arbitrary, ad-hoc command on the test hosts:

```
ansible all -u root -a '/bin/echo This command is run via Ansible'
```

This resulted in the following output:

```
debian8.patch | SUCCESS | rc=0 >>
This command is run via Ansible

debian7.patch | SUCCESS | rc=0 >>
This command is run via Ansible

centos6.patch | SUCCESS | rc=0 >>
This command is run via Ansible
```

Next, the test hosts were grouped by operating system in the hosts file so that group names were defined in square brackets followed by group member hostnames:

```
[CentOS]

centos6.patch


[Debian]

debian7.patch

debian8.patch
```

Grouping was further tested by running an operating system specific update command on servers on both groups:

```
ansible CentOS -u root -a '/usr/bin/yum check-update'

ansible Debian -u root -a '/usr/bin/apt-get update'
```

which resulted in a similar success message with a result listing from the package management software.

Testing automation via Ansible playbooks

After successfully running ad-hoc-commands, the Ansible command language was reviewed by creating a playbook to update the hosts. While ad-hoc commands define exact commands to be run on the target hosts, playbooks define a state the target machine is required to achieve.

A *playbook* is a YAML formatted file consisting of one or more *plays*. Plays in turn contain the target hosts, remote usernames to be used and tasks to be committed. A task includes the core functionality of a playbook: the modules to run with their arguments. For example, the 'service' module can be used to control the state of services – or daemons – on the target hosts or the 'command' module could be used to execute arbitrary commands. The following simple playbook was created in order to test the concept:

```
---
 - hosts: Debian
   remote_user: root
   tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest
    - name: make sure Apache is running
```

```
    service: name=apache2 state=running

- hosts: CentOS

  remote_user: root

  tasks:

   - name: install httpd

     yum: name=httpd update_cache=yes state=latest

   - name: make sure Apache is running

     service: name=httpd state=running
```

In the playbook above, two different plays were defined to achieve the same goals on both Debian and CentOS machines. On both plays the Apache web server is first installed and then it is verified that the service is running. Since Ansible contains built-in core modules for managing apt and yum tools (Ansible Inc (a), 2016, *apt*), they were utilized instead of running arbitrary commands against the targets.

In the first play, Debian host group created earlier is defined as target and the root user is used. Two different tasks are defined, and the installation state of the package apache2 is defined by using the apt module and the state of a service named 'apache2' is defined as 'running'. Similar tasks were defined for the CentOS host group, only using the yum module instead of apt and substituting the apache package and service names with their CentOS equivalents, httpd. The playbook was tested by making sure the Apache service was not installed on any of the target hosts and running the playbook via issuing *ansible-playbook apacherunning.yml*.

The following output was produced:

```
PLAY [Debian]
*************************************************************
*****

GATHERING FACTS
*************************************************************
***
ok: [debian7.patch]
ok: [debian8.patch]

TASK: [install apache2]
*******************************************************
changed: [debian7.patch]
changed: [debian8.patch]

TASK: [make sure Apache is running]
*****************************************
ok: [debian8.patch]
ok: [debian7.patch]

PLAY [CentOS]
*************************************************************
*****

GATHERING FACTS
*************************************************************
***
ok: [centos6.patch]

TASK: [install httpd]
*******************************************************
changed: [centos6.patch]

TASK: [make sure Apache is running]
*****************************************
changed: [centos6.patch]

PLAY RECAP
*************************************************************
********
centos6.patch              : ok=3    changed=2    unreach-
able=0    failed=0
debian7.patch              : ok=3    changed=1    unreach-
able=0    failed=0
debian8.patch              : ok=3    changed=1    unreach-
able=0    failed=0
```

From the listing above it can been seen that the all tasks were successfully executed on the three defined servers. It can also be seen that only a single change was performed on Debian hosts, whereas two changes were required on the CentOS host. This is due to the differences between apt and yum package managers: with apt, an installed service is started upon installation and thus the second task was already at the defined state. In contrast, yum does not start the service. The defined state 'running' was not met, and a change was required with the second task as well. Running the same playbook again results in a slighly different result:

```
PLAY RECAP
*************************************************************
********
centos6.patch              : ok=3    changed=0    unreach-
able=0    failed=0
debian7.patch              : ok=3    changed=0    unreach-
able=0    failed=0
debian8.patch              : ok=3    changed=0    unreach-
able=0    failed=0
```

This time the Apache service was already installed and running on all hosts. The state-based nature of playbooks can easily be observed from the listings above. In the first listing the state did not match and was thus corrected whereas in the latter one the state was already correct and no actions were taken. After verifying the initial playbook functionality in the test environment, a real playbook for upgrading systems was created. The following task was defined for upgrading all hosts utilizing the operating-system specific package manager commands:

```
---
 - hosts: Debian
   remote_user: root
   tasks:
```

```
  - name: Upgrade all packages to latest versions via apt
    apt: upgrade=dist update_cache=yes dpkg_options='force-confold,force-confdef'
- hosts: CentOS
  remote_user: root
  tasks:
  - name: Upgrade all packages to latest versions via yum
    yum: name=* update_cache=yes state=latest
```

Since the Ansible apt module already contained workarounds for interactive dpkg behavior (as experienced in chapter 4.1.5), the relevant dpkg options *force-confold and force-confdef* could be passed to the module. This way possible interactive queries concerning configuration files changes were automatically answered. The playbook was run and all available patches were installed successfully, which was confirmed by checking the situation manually on each host.

Ansible GUI

After successfully running a patching process via Ansible playbooks, a graphical user interface was set up. An official, proprietary web interface Ansible Tower exists, developed by Ansible Inc. However, due to the open-source nature of the study, an open-source alternative – semaphore – was selected as the web UI (User Interface). Semaphore claims to include all the basic features of Ansible Tower, and in addition be simpler to use and have certain additional features such as the ability to run playbooks against specified hosts (Kramny, 2016). The basic idea behind semaphore is simple: running playbooks via a web interface. With semaphore, the playbooks are stored in a git repository and pulled on the semaphore host upon execution of the playbook.

Due to the environmental requirements, semaphore was set up on the Manager host but on a different test host instead. Semaphore was deployed as a Docker container as instructed in the GitHub project page (Kramny, 2016). The required MongoDB and Redis NoSQL databases were installed as separate containers. Since the semaphore container already included Ansible, the included Ansible installation was used. The playbooks created earlier were uploaded to an external git repository. The repository was then defined in semaphore as a 'Playbook'. Since semaphore does not use the Ansible host definitions in /etc/ansible/hosts, the hosts were re-defined in semaphore in a similar fashion, as seen on Figure 10.

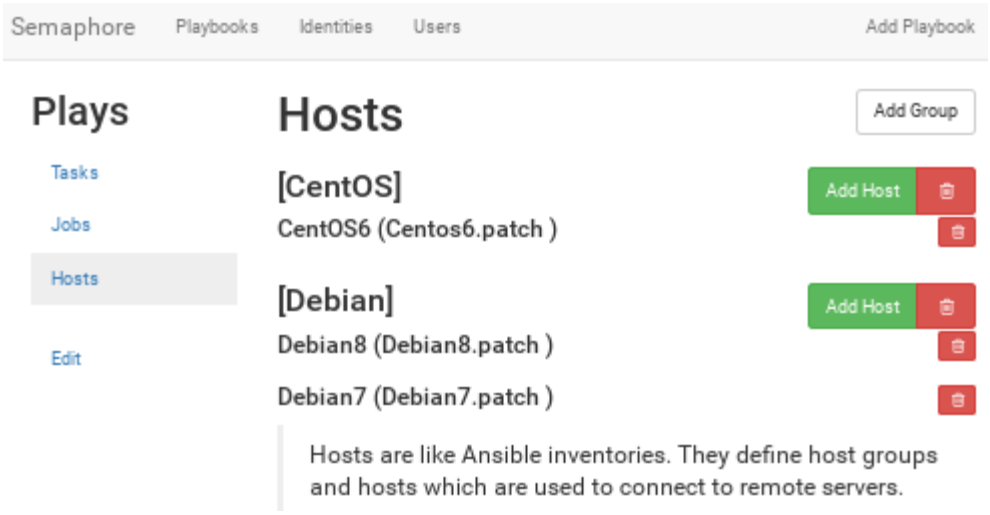

Figure 10. Semaphore hosts definition.

Next, a Job was defined in the semaphore UI. A Job consists of a single play file in the git repository. The same upgrade.yml and apacherunning.yml files as before were introduced in semaphore as different jobs (Figure 11).

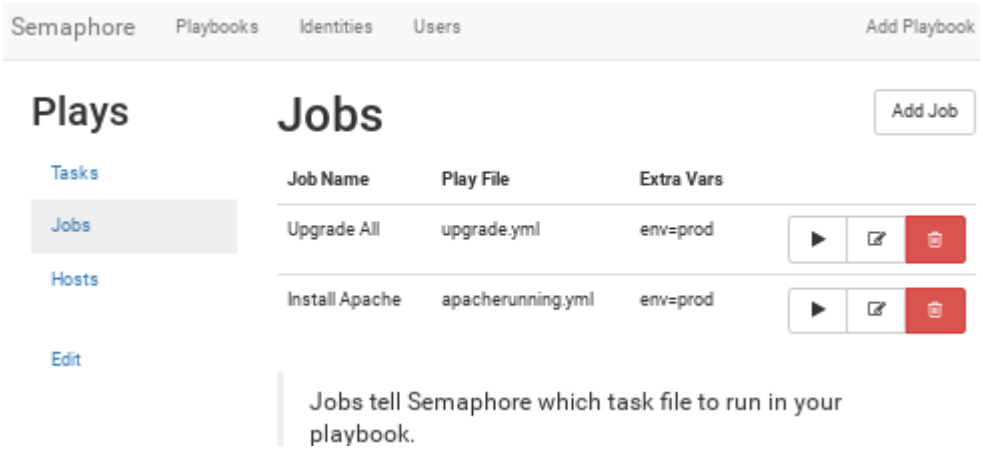

Figure 11. Ansible jobs.

Finally, the jobs were run thus creating 'tasks' in semaphore. The status of the task could be seen on the tasks page and the full output – the same as when running ansible-playbook on the CLI (Command Line) – could be checked via the See Output link as show in Figure 12.

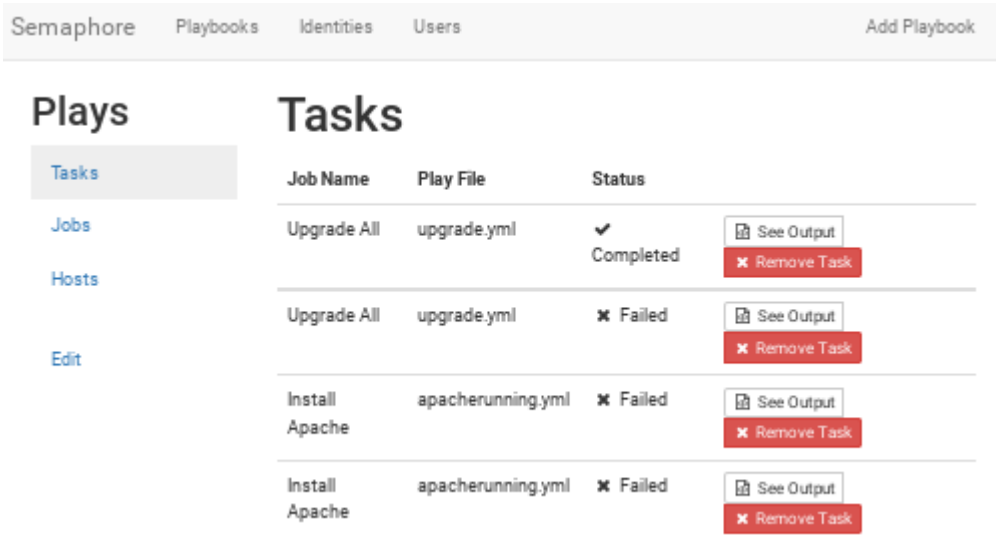

Figure 12. Tasks view in semaphore.

4.3 Patch Management Software

4.3.1 Introduction to Spacewalk

Spacewalk is a Linux systems management solution with the following capabilities (Red Hat Inc. 2015):

- Inventorying systems (e.g. installed software).
- Installing and updating software.
- Provisioning systems.
- Managing and deploying systems configuration.

Hosted and mainly developed by Red Hat Inc, Spacewalk is the open-source, upstream project to the commercial Red Hat Satellite product. Spacewalk acts as a software repository of its own so a separate internal repository as created earlier implementation options was not required. Contrary to pulling updates from a repository such as with the self-created PatchApp tool, in Spacewalk, the updates are pushed from the Spacewalk server to the target hosts. Since the architecture consists of distinct agent software and a management server, Spacewalk is an example of agent-based architecture in NIST lingo (Souppaya, 2013, 8-9).

4.3.2 Installing and Configuring Spacewalk

Spacewalk version 2.3 was installed on the Manager host in the test environment according to the installation howto documentation (The Fedora Project, 2015, *HowToInstall*). In addition, the spacewalk-utils package providing management scripts

etc. was installed. The official repositories were used and PostgreSQL was used as the database backed and the initial setup was run. After completing the installation, the management web interface was up on the Manager host.

Adding Centos hosts

Next, default source repositories were defined for Centos 6 operating system on x86_64 architecture. The corresponding software channels were created as well. The spacewalk-utils package provided a script to create both the repositories and the channels and to link them together:

```
spacewalk-common-channels -v -u <username> -p <password> -a x86_64 -k unlimited
'centos6*' 'spacewalk23-client-centos6'
```

After creating the repositories, one of the channels - the Spacewalk client channel - was synced with the corresponding upstream repository on the Internet using spacewalk-repo-sync script:

```
spacewalk-repo-sync --channel spacewalk23-client-centos6-x86_64
```

At this point there was a functional software channel on the Manager host with Spacewalk client packages for Centos 6 on X86_64. Next, the first managed system, Centos6, was added. The Spacewalk client and EPEL repositories were added on Centos6 and the following packages were installed:

```
rhn-client-tools rhn-check rhn-setup rhnsd m2crypto yum-rhn-plugin
```

The Spacewalk server SSL certificate was installed on the host and the client was registered on the Spacewalk server using the rhnreg_ks tool and the Centos 6 base channel activation key:

```
rhnreg_ks --serverUrl=https://manager.patch/XMLRPC --sslCACert=/usr/share/rhn/RHN-ORG-TRUSTED-SSL-CERT --activationkey=centos6-x86_64
```

The host was now available on the Spacewalk server web UI as shown in Figure 13.

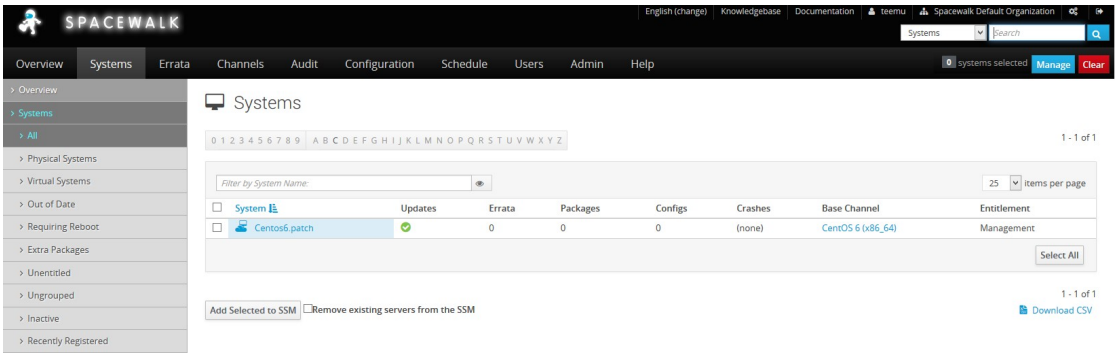

Figure 13. Centos host added to Spacewalk.

Adding Debian hosts

As with Centos, software channels were added for Debian hosts. The Debian repositories typically consist of two different repositories – the default one and a separate ”security” repository and these are both divided into 'main', 'contrib' and 'non-free' sections. In Spacewalk a server can only enlist to a single base channel, though (Coe, 2012). In addition, the hierarchy has only two levels, so sub-sub-channels are not possible. Because of these characteristics, the base channel ”Debian 8 x64” was created to represent Debian Jessie ”main” repository. Sub-channels ”contrib”, ”non-free”, ”security”, ”security-contrib” and ”security-non-free” were created to represent the corresponding Debian repositories (Table 1).

Table 1. Debian repository/channel relations.

Debian repository	Corresponding Spacewalk channel
Debian/main	Debian 8 x64 (base channel for all below)
Debian/contrib	Debian 8 x64 contrib (subchannel)
Debian/non-free	Debian 8 x64 non-free (subchannel)
Debian Security/main	Debian 8 x64 DebSec (subchannel)
Debian Security/contrib	Debian 8 x64 DebSec contrib (subchannel)
Debian Security/non-free	Debian 8 x64 DebSec non-free (subchannel)

An activation key *1-debian8* was created for the base channel and automatically propagated to all its child channels as well. Since Debian is not as well supported within the Spacewalk environment as the Red Hat derivative distributions, it was not possible to populate the repositories by defining an upstream repository within Spacewalk itself. Instead, the channels were populated using Meier's (2013) spacewalk-debian-sync script. The script was given the Spacewalk credentials, the Spacewalk channel label and the upstream repository URL:

```
./spacewalk-debian-sync.pl --username admin --password 'XXX' --channel
'debian_6_x64_lts' –url 'http://ftp.fi.debian.org/debian/dists/squeeze-lts/main/binary-amd64/'
```

Installing Perl module WWW:Mechanize and copying the Spacewalk certificate from /var/www/html/pub/RHN-ORG-TRUSTED-SSL-CERT to /usr/share/rhn/RHN-ORG-TRUSTED-SSL-CERT was required to make the script work. After that, the script immediately began pulling packages from the defined Finnish Debian repository mirror to the local host as displayed in Figure 14.

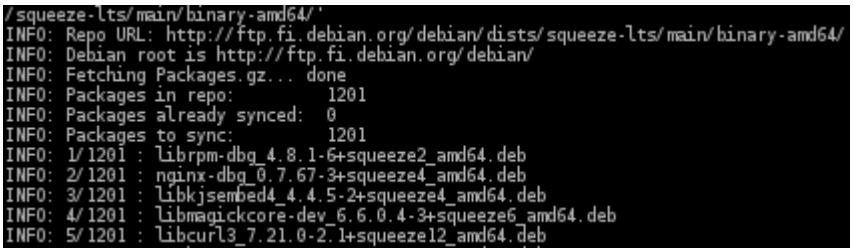

Figure 14. Populating Spacewalk repository manually

Client packages *apt-transport-spacewalk* and *rhnsd* were installed on Debian hosts to provide Spacewalk client capabilities. The systems were then registered to Spacewalk with rhnreg_ks utility using the newly created 1-debian8-key. Based on the key, the base channel was set automatically by rhnreg_ks, and the subchannels were added manually in Spacewalk administration interface. The Debian host was now successfully added along with the CentOS host.

Demonstrating Spacewalk Functionality

After adding the managed hosts to Spacewalk, their status, such as missing patches count, could be easily perceived on the web interface as seen in Figure 15.

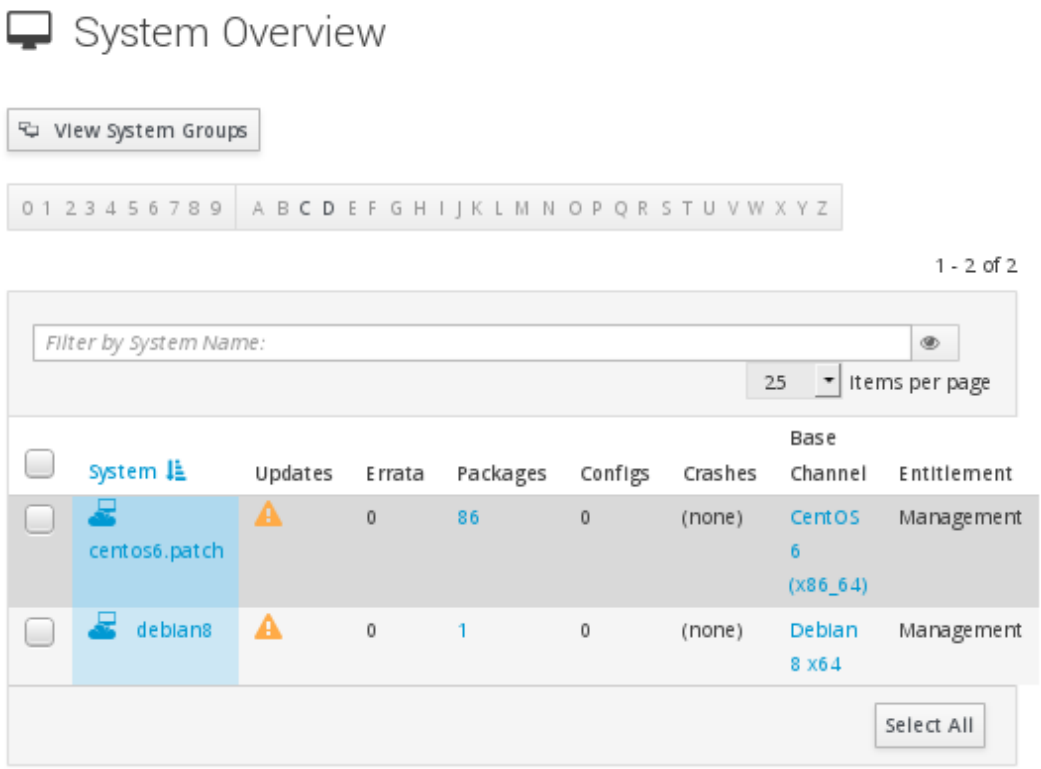

Figure 15. Spacewalk hosts view.

The hosts could trivially be patched as well, either completely or by package (Figure 16).

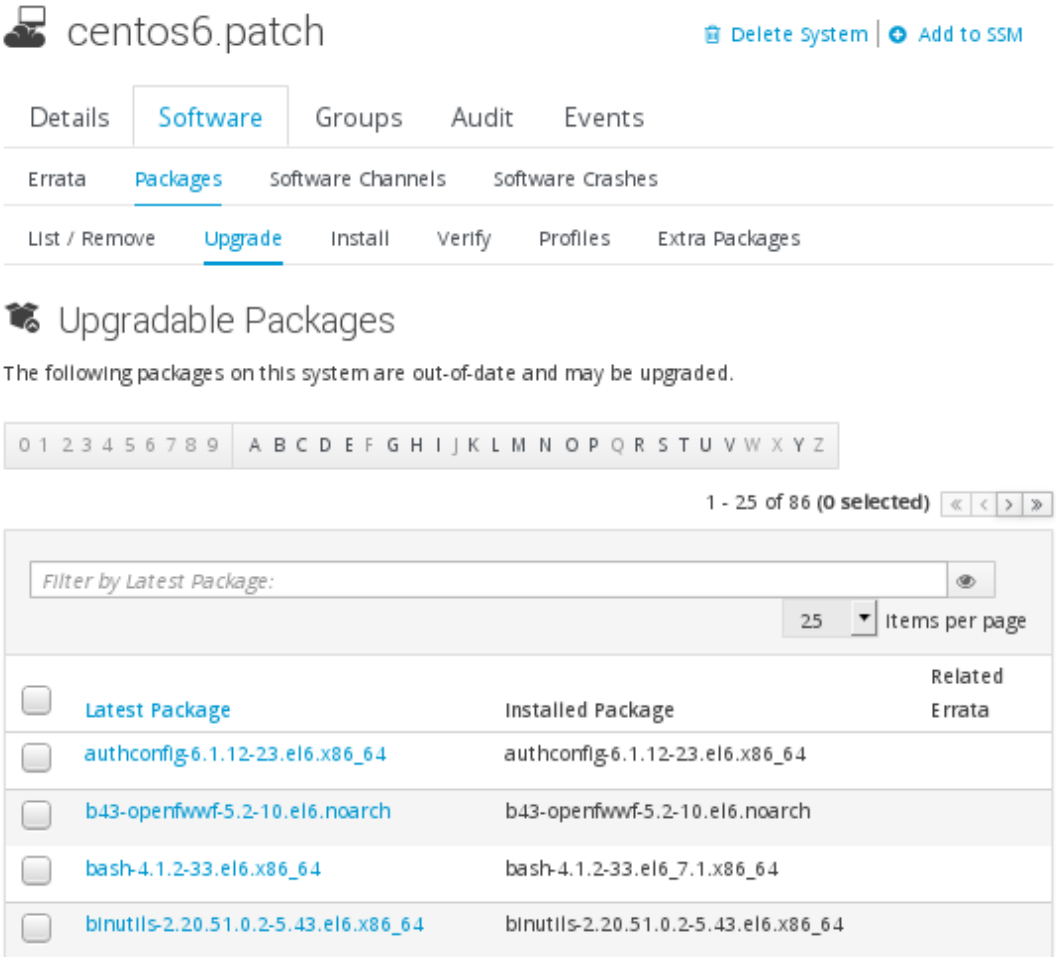

Figure 16. Spacewalk upgrade view.

4.4 Comparison

The goal of comparing the implemented solutions was to determine how each solution fulfills the organization's requirements and how it compares to the best practices in the literature. A holistic approach was taken – as befits a case study – instead of finding an existing comparison method. Knowledge gathered in the implementation phase was heavily used when determining the capabilities of each approach. In order to equally compare the solutions examined in previous chapters, a set of features were defined. The following features were conducted from the organization requirements described in Chapter 1.4:

- Ease of use (emphasis on experience by people with little Linux know-how)
- Functionality (ability to perform actual update/management tasks)
- Reportability (ability to produce reports, show errata etc.)
- Maintainability (future of the solution, availability of support, ease of updating etc.)

Based on the experiences in Chapter 4, each solution was verbally reviewed per feature (Table 2).

Table 2. Verbal review.

	MLOY PatchApp	Ansible	Spacewalk
Ease of Use	Web app, clumsy UI	Web app, feature-incomplete UI	**Web app, professional UI**
Functionality	OS Patching, easily extended to application patching	OS patching, host management, easily extended to application patching	**OS Patching, Scheduling, Asset Inventory, Per-package updates,**
Reportability	Installed packages only	Installed packages only	**Installed packages, missing patches, security errata, host information**
Maintainability	Single maintainer, do-it-yourself, bad architecture	Open source community	**Backed by Red Hat, open source community, paid support available. Future plans unknown**

Emphasis factors – represented by integers – were given to different features based on importance of each feature in the assigner's environment. The factors were defined as follows:

- Ease of use: 3
- Functionality: 3
- Reportability: 1
- Maintainability: 2

Based on the verbal review, each solution was graded on a scale from one to ten regarding each of the features. The final grade was calculated by multiplying the scores with the corresponding factor and finally summing the points together. (Table 3).

Table 3. Calculating the final scores.

Feature (factor)	MLOY PatchApp Score (weighed)	Ansible Score (weighed)	Spacewalk Score (weighed)
Ease of Use (3)	6 (18)	7 (21)	**8 (24)**
Functionality (3)	5 (15)	6 (18)	**8 (24)**
Reportability (1)	3 (3)	3 (3)	**8 (8)**
Maintainability (2)	5 (10)	7 (14)	**8 (16)**
Weighed sum of the above	**46**	**56**	**74**

Based on the weighed sum, Spacewalk was found to be the recommended solution for the assigner's environment.

5 Conclusions

The results achieved with each solution are discussed in more detail in the following chapters and conclusions are made based on the results. The positive and negative aspects are weighed and the suitability for different environments by approach is reflected on. Final conclusions are made and the thesis process is also reflected on.

5.1 Self-Created PatchApp

It was demonstrated in Chapter 4.1 that a functional patch management implementation could indeed be put together from scratch by utilizing native operating system

functionality remotely over SSH and deploying a web interface on top of it. The application does cover several of the requirements defined in the Objectives and Methods chapter as well as those presented in literature such as in White's proposition (2004). For instance, a web interface for simple patch deployment exists, as well as the capability to operate multiple operating systems. No distinct method of handling vulnerability notifications was defined, however. The notifications should instead be handled by subscribing to vendor-specific and general security mailing lists as suggested by White (2004, 4). The “pulling patches” category in White's model (2004, 9) is obviously implemented as well, in the form of using the official distribution repositories. The “various sources” issue presented by White (2004, 9) is handled by using an internal repository. In contrast to White's proposal (2004, 8), most of the detection and testing functionality is missing. While it is possible to see the situation of the current available updates, other than that, reporting functionality does not really exist either. The existing functionality is also somewhat awkward: for instance the web interface waits for an update command to finish before showing the results.

There is certainly potential in the self-created approach, and since it is built from the ground up, there are practically no limits functionality-wise. As an agentless approach (Souppaya, 2013, 8-9), it requires little modification on the target hosts, which is often beneficial in production environments. It could easily be extended to application patching, for instance. Whereas operating system upgrades are usually simple to implement using native operating system tools, different third-party applications can be quite a different thing, though. The applications have different installation and update methods which may or may not integrate to the native package management system, which would re-introduce the mentioned White's (2004, 9) “various sources” issue. The installation itself can usually be scripted and such scripts are trivial to call on a web interface such as the one implemented here.

Only a limited feature set exists due to time constraints and lack of software development skills. Some architectural choices were not too desirable, such as calling Bash scripts from within Python, instead of utilizing Python libraries for the same task. The self-created approach is also self-maintained. An organization developing an entirely custom solution might find maintenance resource-intensive. Even so, a custom solution might be a viable approach for heavily customized, diverse environments where development workforce exists within the organization.

5.2 Configuration Management Approach

Configuration management software is an established way of controlling systems and it goes without saying that such software can be utilized for patch management as well, as demonstrated in Chapter 4.2. Many of the difficulties found with the self-created solution – such as the web interface responsiveness and handling configurations file changes were already addressed in Ansible. With Ansible it was also possible to control which hosts to patch via the web interface, but as with PatchApp, individual patches were not easily handled. Other than that, the results were somewhat similar to the self-created PatchApp solution. A web interface was present as well and the result was vendor-neutral. The solution was also agentless (Souppaya, 2013, 8-9). Ansible, just like PatchApp, utilizes operating system tools, so the only package update option is replacing binaries instead of patching the compiled binaries (White 2004, 8-9). There was little reporting functionality and no built-in security errata information. It should be noted, however, that most of the limitations apply to the implemented solution, not configuration management as a whole.

Obviously, results with configuration management software may vary dramatically depending on the environment. An organization with an existing configuration management solution is likely to prefer that for patch management as well. As briefly in-

troduced in Chapter 4.2, there are numerous different configuration management tools available and some of them might be more applicable for patch management.

5.3 Spacewalk

As anticipated, Spacewalk – being a dedicated patch management tool – was the most streamlined solution for patch management out of the box. In addition to what was achieved with the other solutions, Spacewalk offered per-package management, scheduling and a host of reporting possibilities such as status information. Information gathering was also covered in the Spacewalk implementation and a systems database was automatically created when adding hosts, as suggested by both Nicastro (2011, Chapter 2) and Souppaya (2003, 9). System information was readily available, and the published updates could be tracked via errata information. While errata is not available by default for the open source operating systems and errata collection was not implemented in the example setup, implementing errata by collecting data from vendor announcement emails with existing scripts should be a simple task (Meier, 2016). Errata information could be used for classifying the vulnerabilities and scheduling patches based on this information as Nicastro suggests (2011, Chapter 9).

As an agent-based system (Souppaya, 2013, 8-9) Spacewalk does require software installation on the target hosts, though, which is hardly desirable in production environments. It is also worth mentioning that the future of Spacewalk is somewhat uncertain. While Spacewalk is an upstream project for Red Hat Satellite 5, Satellite 6 already uses different projects as its base. Spacewalk will thus probably be abandoned by Red Hat in the future. Spacewalk is, however, an open source project with contributors other than Red Hat so it could well live on even without Red Hat support. (Red Hat Inc. 2015. Spacewalk FAQ).

5.4 Final Conclusions

During the study it became evident that a patch management system could indeed be implemented using any of the methods presented in Chapter 4. There hardly is a solution that is best for every use case. The required functionalities, the tolerated amount of work and the existing environment as well as the level of operator expertise should be considered when selecting an approach for patch management. All of the options introduced in this study include some form of a graphical user interface, for example. A GUI might, however not be required at all in many environments, or even be a hindrance. Likewise, if the company workflow is heavily dependent on configuration management software, utilizing the same software for patch management could be the best option. Similarly if an environment consists solely of a single vendor operating systems, a vendor-specific approach could be the simplest choice.

Certain characteristics of each option were identified. A fully self-created solution is the most flexible: in its simplest form, one-liner scripts could be run from an arbitrary host to the patched hosts requiring practically zero overhead. Obviously very complex scenarios could be covered as well, however, this would probably require an extensive amount of work. A configuration management system approach is usually very flexible as well since the tools are planned to answer to very different needs. The example solution with Ansible and semaphore is actually very close to the self-implemented example option. The key point when considering a configuration management software really is whether such a software is already used or is planned to be used in the future. Implementing a configuration management setup solely for patch management purposes might not be worth the effort. Not surprisingly, a dedicated patch management software does answer the patch management requirements most closely with quite a small effort. However, a somewhat heavy infrastructure is required and its usefulness outside patch management is very limited.

The research could be used as a basis on implementing a patch management system in any organization, but might particularly be of use in the following scenarios:

- The environment is heterogeneous, consisting of multiple operating systems. The solutions presented in this study can be used in most Linux systems and to some extent in other systems as well. Many commercial systems are more limited in operating system support.
- The organization is unable to invest a large sum of money. All of the solutions presented are fully open source, free of any licensing costs, as required by White (2004, 6). Personnel expenses caused by implementation should naturally be considered nonetheless.

None of the options offer all of the components described in literature, such as in White's (2004) patch management system proposal. That was not the point either, since the theoretical models often handle processes as well as systems, which was beyond the scope of this research. All of the Whatever the selected approach is, some form of a patch management system can be considered a requirement to address the threats presented by a huge number of public vulnerabilities discovered each month.

5.5 Reflections on the Research and Further Study

The research as a whole was rewarding, yet time-consuming since there was little dedicated time for just conducting the research. The research offered more experience in project management and obviously in technical details concerning the topic as well. Some of the topics covered by the degree programme could be utilized, such

as the PKI infrastructure. As a whole, however, the research did not have too much in common with the degree programme, which had both negative and positive effects. On one hand, the research was refreshingly different compared to the other studies, but on the other hand the programme did not give much background in conducting the research. The achieved results are somewhat subjective, but still have a basis on literature and commissioner requirements. This is emphasized in the report and is also natural for a case study.

In hindsight, a more solid timetable should have been created. The research questions should have been more clearly defined already on the beginning. A case study as a research method gave plenty of liberties, however, little structure for the thesis. In might have been beneficial to narrow the subject even more as well, for example by concentrating solely on configuration management. The lack of programming experience was clearly seen when conducting the from scratch -approach. Consultation regarding Django programming best practices would have been helpful. Some kind of a framework for doing the comparison could have been beneficial as well. Most similar reports, however, also develop a comparison criteria of their own.

Further research could include a more in-depth view on some of the implementations. Configuration management could also be studied in more detail, without limiting to patch management functionality. The research consisted solely of self-created and open-source solutions, thus commercial alternatives for patch management could be researched to mirror the results.

References

Maatalouden Laskentakeskus Oy. 2014. *Vuosikertomus 2013*. Book. Vaasa: Oy FRAM Ab.

Ansible Inc (a). 2016. *Ansible Documentation*. Web site. Accessed on March 17th 2016. Retrieved from http://docs.ansible.com/ansible/.

Ansible Inc (b). 2016. *How Ansible Works*. Web page. Accessed on March 17th 2016. Retrieved from https://www.ansible.com/how-ansible-works.

Ansible Inc (c). 2016. *Use Case: Configuration Management. W*eb page. Accessed on March 17th 2016. Retrieved from http://www.ansible.com/configuration-management.

Aro, J. 2014. "*Heartbleed altistanut valtavan määrän dataa hakkereille – asiantuntijat suosittelevat kaikkien salasanojen vaihtoa".* Accessed on May 4th, 2016. Retrieved from http://yle.fi/uutiset/heartbleed_altistanut_valtavan_maaran_dataa_hakkereille__asiantuntijat_suosittelevat_kaikkien_salasanojen_vaihtoa/7181018. Yleisradio.

Becker B. et al. 2012. *Case Studies*. Guide. Accessed on April 13, 2016. Retrieved from http://writing.colostate.edu/guides/guide.cfm. Colorado State University.

Codenomicon. 2014. *The Heartbleed* bug. Codenomicon Ltd. Blog post. Accessed on February 2nd 2015. Retrieved from http://heartbleed.com/

Coe, C. 2012. *Re: [Spacewalk-list] adding multiple base channels to a system.* Email message. Accessed on March 17th 2016. Retrieved from https://www.redhat.com/archives/spacewalk-list/2012-June/msg00228.html. Spacewalk mailing list.

Dreyfuss, J. 2015. *Deployment Management Tools: Chef vs. Puppet vs. Ansible.* Blog post. Accessed on March 17th 2016. Retrieved from http://blog.takipi.com/deployment-management-tools-chef-vs-puppet-vs-ansible-vs-saltstack-vs-fabric/

Ermishkin, N. et al. 2016. "*ImageMagick Is On Fire — CVE-2016–3714".* Accessed on May 4th, 2016. Retrieved from https://imagetragick.com/.

Hertzog , R et al. 2016. *The Debian Administrator's Handbook*. Accessed on March 17th 2016. Retrieved from http://debian-handbook.info.

Jang, M. 2006. *Linux Patch Management.* Prentice Hall. Book. Accessed on January 28th 2015. Retrieved from http://ptgmedia.pearsoncmg.com/images/9780132366755/downloads/0132366754_Jang_book.pdf. Pearson Higher Education.

Jonassen Hass A. 2003. *Configuration Management Principles and Practice*. Book. Accessed on March 17th 2016. Retrieved from http://www.pearsonhighered.com/samplechapter/0321117662.pdf

Kramny, M. et al. 2016. *Semaphore - Open Source Alternative to Ansible Tower.* Readme file. Accessed on March 17th 2016. Retrieved from https://github.com/ansible-semaphore/semaphore

McNabb, A. et al. 2015. *Parallel SSH*. Readme documentation. Accessed on March 17th 2016. Retrieved from https://code.google.com/p/parallel-ssh/

Meier, S. 2013. *spacewalk-repo-sync.* Github software repository. Accessed on March 17th 2016. Retrieved from https://github.com/stevemeier/spacewalk-debian-sync/

Meier, S. 2016. *CEFS: CentOS Errata for Spacewalk.* Web article. Accessed on May 5th 2016. Retrieved from https://cefs.steve-meier.de/.

Nicastro F. 2011. *Security Patch Management, Second Edition*. Auerbach Publications. Book. Accessed on January 30th. 2016. Retrieved from http://library.books24x7.com/toc.aspx?bookid=47178.

NVD. 2014. *Vulnerability Summary for CVE-2014-7169*. National Institute of Standards and Tecnology, National Vulnerability Database. Summary. Accessed on February 6th 2015. Retrieved from http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7169.

Open Source Vulnerability Database. 2016. *OSVDB: FIN.* Blog post. Accessed on April 11th 2016. Retrieved from https://blog.osvdb.org/author/jerichoattrition/.

Parallel SSH issue tracker. 2013. *parallel-ssh*. Accessed on March 17th 2016. Retrieved from https://code.google.com/archive/p/parallel-ssh/issues. Google Code.

Puppet Labs 2016. *Introduction to Puppet.* Web article. Accessed on March 17th 2016. Retrieved from https://docs.puppetlabs.com/guides/introduction.html.

Puppet Labs Inc. 2015. *What is Puppet*. Web article. Accessed on March 17th 2016. Retrieved from https://puppetlabs.com/puppet/what-is-puppet.

Red Hat Inc. 2015. *Spacewalk documentation*. Web site. Accessed on March 17th 2016. Retrieved from http://spacewalk.redhat.com/.

Sarwate, A. 2015. *The GHOST Vulnerability.* Qualys Inc. Blog post. Accessed on February 6th 2015. Retrieved from https://community.qualys.com/blogs/laws-of-vulnerabilities/2015/01/27/the-ghost-vulnerability.

Shuttleworth, M. 2008. *Case Study Research Design*. Article. Accessed on April 13, 2016. Retrieved from https://explorable.com/case-study-research-design. Explorable.com.

Souppaya, M et al. 2013. *Guide to Enterprise Patch*

The Fedora Project. 2015. *Spacewalk Installation Instructions*. Wiki page. Accessed on March 17th 2016. Retrieved from https://fedorahosted.org/spacewalk/wiki/HowToInstall.

Voldal, D. 2003. *A Practical Methodology for Implementing a Patch Management Process.* Sans Institute. White Paper. Accessed on January 30th. 2016. Retrieved from

https://www.sans.org/reading-room/whitepapers/bestprac/practical-methodology-implementing-patch-management-process-1206

White D. 2004. *A Unified Architechture for Automatic Software Updates.* Rhodes University. Paper. Accessed on January 30th. 2016. Retrieved from http://icsa.cs.up-.ac.za/issa/2004/Proceedings/Full/024.pdf.

Appendices

Appendix 1. listUpdates HTML template.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
 <head>
     <meta charset="UTF-8" />
     <title>MLOY Patching</title>
     <link rel="stylesheet" href="/static/PatchWeb.css" type="text/css" />
 </head>

 <body>
     <div id="header">
         <IMG src="/static/mtlk-logo.png" align=left>
         <H1> MLOY Patching app </H1>
     </div>
     <div id="menu">
         <a href="/">Front page</a> <br />

         <a href="listUpdates.html">Show available updates</a> <br />

         <a href="runAllUpdates.html" onclick="return confirm('The requested action will be run without further confirmation! Are you sure?')" style="color:#FF0000 !important" >Run all updates</a> <br />
```

```
            <a href="mailto:teemu@tmu.fi">Email admin</a><br />
        <br />
         </div>
         <div id="data"> <br />
            <h2 style="text-align: center;">Packages available for upgrade</h2>
            <br />
            <br /> <pre>{{ output }}</pre>
         </div>
    </body>
</html>
```

Appendix 2. PatchWeb css file.

```
html
{min-height:100%;}

body
{height:100%;}

#menu a
{
    display:block;
    width:110px;
    padding:5px 18px 5px 0;
    color:#606060;
    background:#e0e0e0;
    font-size:1.8em;
```

```
    font-weight:normal;
    text-decoration:none;
    letter-spacing:-2px;
}

#menu
{
    width:130px;
    height:1300px;
    margin-left:20px;
    margin-right: 20px;
    float:left;
    background-color:#EEEEEE;
}

#data
{
    overflow:auto;
    background-color:#000000;
    color:#00FF00;
}

#header
{
    width:100%;
```

```
        height:130px;
        float:top;
        background-color:#EEEEEE;
        text-align:center;
        margin-bottom: 20px;
}
```